Infix Operaatorid I

- Infix operaatorid (näiteks +) ja tüübid (näiteks ->) kirjutatakse argumentide vahele, mitte argumentide ette.
  - Näiteks: `5 + 2`, `2*pi*r^2`, `Float -> Int`
- Infixoperaatori kasutamiseks prefix-vormis tuleb see panna sulgudesse. Näiteks: `3 + 4 == (+) 3 4`
- Infixoperaatoreid saab ise juurde defineerida:

```
(@) :: Int -> Int -> Int
x @ y = 2*x+y
```

või

```
(@) :: Int -> Int -> Int
(@) x y = 2*x+y
```

Infix Operaatorid II

- Infixoperaatoril on prioriteet (i.k. precedence)
  - Näiteks: `3*4+5 == (3*4)+5`, kuna `*` on tasemel 7 ja `+` tasemel 6
- Infixoperaatoril võib olla assotsiatiivsus
  - näiteks, `2**3**4 == 2**(3**4)`, `2 - 3 - 4 == (2-3)-4`
- Neid seatakse nn. fixity deklaratsioonidega:

```
infixl 7 *
infixl 6 +
infixl 6 -
infixr 8 **
```

- Fixity-deklaratsioone saab vaadata ghci abil:
  - Näiteks käsuga “:i (+)”
- Prefix-operaatoreid saab rakendada infix-selt tagurpidi-ülakomade (`) vahel
  - Näiteks: `5 `div` 2`

Eeldefineeritud operaatorite fixity (Haskell 98)

Prec-edence	Left associative operators	Non-associative operators	Right associative operators
9	!!		.
8			^, `l` **
7	*, /, ‘div‘, ‘mod‘, ‘rem‘, ‘quot‘		
6	+, -		
5			:, ++
4		==, /=, <, <=, >, >=, ‘elem‘, ‘notElem‘	
3			&&
2			||
1	>>, >>=		
0			$, $!, ‘seq‘

Unit tüüp ja ennikud

Kõige triviaalsem väärtus Haskellis on `()`.

- Selle tüüp on samuti `()` ehk `() :: ()`
- Kasutatakse juhtudel, kui pole vaja informatsiooni edastada.
- Paljudes keeltes kasutatakse tüüpi `void`

Olemasolevatest tüüpidest saab luua paare ja muid ennikuid:

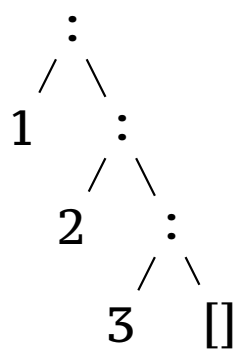
- `(1,'a',()), ((1.1,8),False)`
- enniku tüüp konstrueeritakse komponentide tüüpidest
  `(1,'a',()) :: (Int,Char,())`
- paarides olevat info saab kätte mustrisobitusega:

```
f :: (Int, Char, ()) -> Int
f (x,c,()) = x + 1
```

Järjendid e. listid

- Järjendi tüübiks on “[a]”, kus “a” on järjendi elementide tüüp.
  - Näiteks: `[Int]`, `[Char]`, `[[Float]]`
- Järjendeid saab kirjutada nii (tüüpide kirjutamine vabatahtlik):
  - `[1, 2, 3] :: [Int]`, `[3] :: [Int]`, `[] :: [Int]`
  - `[True, False, False] :: [Bool]`, `[False] :: [Bool]`, `[] :: [Bool]`
- Haskelli listid on arvuti mälus esindatud puudena:

`[1, 2, 3] ==`



- Järjendi loomiseks on kaks konstruktorit, mis vastavad (list-tüüpi) puu tippudele.
  - `[] :: [a]`
  - `(:) :: a -> [a] -> [a]`
  - Näide: `[3,2,1] == 3 : 2 : 1 : []`

Järjendi e. listid

- Konstrueerimise vastand on mustrisobitus. Nii saame vaadata, millise konstruktoriga antud väärtus loodi.

```
length []     = 0
length (x:xs) = 1 + length xs
```

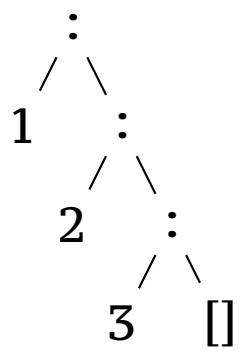
või

```
length xs =
  case xs of
    []     -> 0
    (x:xs) -> 1 + length xs
```

- Mustreid sobitatakse "ülevalt all", kuni esimese sobiva leidmiseni!
- Arvutuskäik:

```
length [1,2,3]
==> length (1 : (2 : (3 : [])))
==> 1 + length (2 : (3 : []))
==> 1 + (1 + length (3 : []))
==> 1 + (1 + (1 + length []))
==> 1 + (1 + (1 + 0))
==> 3
```

Tähelepanekud



- Elemendi lisamine listi algusse väga kiire!
  - kiire == konstante aeg ja mälu
  - Põhjendus: argumente ei ole vaja kopeerida.
- Viimase elemendi selekteerimine aeglane!
  - aeglane == lineaarselt listi pikkusega
- Järjendi lõppu lisamine aeglane!
  - aeglane == aeg ja mälu lineaarne listi pikkusega (tuleb kopeerida)
  - Laiks väärtustamine päästab mõnel juhul!
- $\Longrightarrow$ Listid Haskellis on kasutatavad iteraatoritena

Loe lisaks: RWH, lk 10..12, 23..26; LYaH, peatükk 2 lõpp

Laisk väärtustamine I

programm: rida deklaratsioone. Näiteks:

```
double x = x + x
main = double (1+1)
```

redex: redutseeritav avaldis – programmis olev avaldis, mida saab lihtsustada. Näiteks: `1+1`

kontekst: kõik definitsioonid, mis kehtivad redex-i asukohas.

Näited ülevaloleva programmi kohta:

- `1+1` puhul on kontekstis kogu programm, kuid neid definitsioone ei lähe vaja.
- järgmisel sammul, kui redexiks `x + x` on selle näite puhul kontekstis ka `x = 2`
- `double (1+1)` puhul on kontekstis kogu programm, kuid vaja läheb vaid `double` definitsiooni.

Laisk väärtustamine II

Lihtsustada saab

1. sisseehitatud operaatoreid, aga ainult siis, kui argumendid on juba normaalkujul.
   - `1 + x` ei saa lihtsustada
   - `1 + 1` lihtsustub avaldiseks `2`
2. nimesid, mis on kontekstis defineeritud
   - `x` lihtsustub arvuks `5`, kui kontekstis on `x = 5`
3. argumendi rakendamist lambda-avaldisele
   - `(\x -> x+x) 5` lihtsustub avaldiseks `x+x where x = 5`
4. funktsioonirakendust

   $f\ e_1\ \ldots\ e_n$ (kus $f\ x_1\ \ldots\ x_n = b$)

   $\Downarrow$

   b **where** { $x_n = e_n$; $\ldots$ $x_n = e_n$ }

   Näiteks: `double (1+1)` lihtsustub avaldiseks `x+x where x = (1+1)`

- Avaldis on *normaalkujul*, kui teda ei saa enam lihtsustada!
- Kui kontekstis on juba muutuja defineeritud, tuleb funktsiooni/lambda argument ümbernimetada!

Redutseerimise järjekord I

Redutseerimiseks valida kõige välimisem avaldis. Kui välist avaldist ei saa redutseerida, võtame ette selle alamavaldised, suunaga vasakult-paremale.

Programm:

```
double x = x + x
main = double (1+1)
```

Reduktsioon (nn. laisk väärtustamine, Haskell):

```
main
==> double (1+1)
==> x + x where x = 1+1
==> x + x where x = 2
==> 2 + 2
==> 4
```

Redutseerimise järjekord II

Kui mitte kasutada *where*-konstruktsiooni, tekivad sisse kordused!

Programm:

```
double x = x + x
main = double (1+1)
```

Reduktsioon (nn. normaaljärjekord):

```
main
==> double (1+1)
==> (1+1) + (1+1)
==> 2 + (1+1)
==> 2 + 2
==> 4
```

Redutseerimise järjekord III

Normaalkujul või ühekordselt esinevad argumendid võime kohe asendame funktsiooni! (S.t. ei pea lisama *where*-konstruktsiooni.)

Programm:

```
fact 0 = 1
fact x = x * fact (x-1)
main = fact 2
```

NB! Kui lihtsustamist takistab mustrisobitus, tuleb redutseerida sobitatavat avaldist (nii vähe kui võimalik)!
Reduktsioon:

```
main
==> fact 2
==> 2 * fact (2-1)
==> 2 * fact 1
==> 2 * (1 * fact 0)
==> 2 * (1 * 1)
==> 2 * 1
==> 2
```

Loe lisaks: RWH, lk 32..36

Kõrgemat järku funktsioon I

… on funktsioon, mis võtab argumendiks (või tagastab) funktsiooni. Näiteks

```
map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (x:xs) = f x : map f xs
```

Funktsiooni `map` illustreerib järgmine võrdus:

$$\texttt{map}\ f\ [x_1, x_2, \ldots, x_n] = [f\ x_1, f\ x_2, \ldots, f\ x_n]$$

FP keskne mõte: arvutada saab ka arvutustega (e. funktsioonidega).

Näiteks `map` saab interpreteerida kui `(a -> b) -> ([a] -> [b])`

Kõrgemat järku funktsioon II

Näiteks

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f b []     = b
foldr f b (x:xs) = f x (foldr f b xs)
```

Funktsiooni `foldr` illustreerib järgmine võrdus:

$$\texttt{foldr}\ (+)\ b\ [x_1, x_2, \ldots, x_n] = x_1 + (x_2 + (\ldots + (x_n + b)))$$

Funktsiooni `map` saab defineerida läbi `foldr`-i

```
map f xs = foldr g [] xs
   where g x y = (f x) : y
```

ehk

$$\texttt{foldr}\ g\ []\ [x_1, x_2, \ldots, x_n] = x_1 `g` (x_2 `g` (\ldots `g` (x_n `g` [])))) = f\ x_1 : f\ x_2 : \ldots : f\ x_n : []$$

Kõrgemat järku funktsioon III

Näiteks

```
foldl :: (b -> a -> b) -> b -> [a] -> b
foldl f b []     = b
foldl f b (x:xs) = foldr f (f b x) xs
```

Funktsiooni `foldl` illustreerib järgmine võrdus:

$$\texttt{foldl}\ (+)\ b\ [x_1, x_2, \ldots, x_n] = (((b + x_1) + x_2) + \ldots) + x_n$$

Funktsiooni `reverse` saab defineerida läbi `foldl`-i

```
reverse xs = foldl g [] xs
  where g x y = y : x
```

ehk

$$\texttt{foldl}\ g\ []\ [x_1, x_2, \ldots, x_n] = ((([]\ `g`\ x_1)\ `g`\ x_2)\ `g`\ \ldots)\ `g`\ x_n = x_n : \ldots : x_2 : x_1 : []$$

Miks eelistada kõrgemat järku funktsioone

- Programmeerijatena ei jõua me kaugele mõeldes *bittide*-tasemel.
- Peame jõudma kõrgemale tasemele — abstraktsemalt

```
f :: [Double] -> [Double]
f (0:_)  = []
f (x:xs) = x + f xs
f []     = []
```

vs.

```
f = sum . takeWhile (/=0)
```

Loe lisaks: RWH, peatükk 4, lk 84..99

Haskelli tüübisüsteem

Haskelli on *staatiliselt tüübitud*, *tugevalt tüübitud*, *tüübituletusega* programmeerimiskeel.

- staatiliselt tüübitud — tüübid teada kompileerimise ajal
- tugevalt tüübitud
  - garantii, et väärtused on just seda tüüpi, millega nad end esitavad
  - tüüpe ei teisendata automaatselt. (nagu C-s int -> float)

S.t. rohkem eeltööd, et tüübivead eemaldada. Programm on aga siis töökindlam.

- tüübituletus — enamasti kirjutatakse tüüpe ainult selleks, et kontrollida, kas kompilaator saab programmist samamoodi aru nagu programmeerija.